`typedef` : típusnév definiálása

`typedef` áll elöl, utána definíció, melyben az a név, amely a változó, vagy függvény neve helyén áll, a típus neve lesz.

Pl.:

`typedef int *ip_t;`	:`ip_t` típusnév lett, nem változó
`ip_t g, ipt[10];`	:g típusa `int*`, `ipt`: 10 elemű, `int`-re mutató pointerekből álló tömb
`typedef int f_t (double, char);`	:`f_t` függvény típus neve
`void vf (f_t p1, int p2);`	:OK; `p1` egy fv. paraméter
`f_t f1 { .... }`	:HIBA: függvény definíciót nem lehet típusnévvel megadni
`f_t f2;`	: OK.: ez függvény deklaráció

Több indexű tömbök

A több dimenziós tömbök igazából tömbök tömbjei, pl:

```
double dt [10] [20];        :  2-indexű, nem  dt [10,20]
int t3 [5][44][10];         :  3-indexű (akárhány index lehet)
double * dp;
        ...
dt[2][3-z] = 34.56;         :  ez sem írható így:   dt[2,3-z]
dp=dt[5];                   :  dt  5. sorának címét teszi dp-be,
```

mert ha csak az első indexet adjuk meg, az eggyel kevesebb indexű tömböt ad; itt éppen 1-indexűt

A memóriában a több indexű tömb utolsó indexe változik a leggyorsabban, azaz pl. 2-indexű tömb sorfolytonosan tárolódik:

```
dt[0][0]  dt[0][1]   ...  dt[0][19]  dt[1][0]  dt[1][1]
...
```

így tehát pl. `dt[0][20]` ≡ `dt[1][0]`

```
t3[0][0][0]  t3[0][0][1]  ...  t3[0][0][9]  t3[0][1][0]
t3[0][1][1]  ... t3[0][1][9]  t3[0][2][0] ...
t[30][2][9]
... t3 [0][43][9]  t3 [1][0][0]  ...  t3 [4][43][9]
```

Függvény paraméterénél az első indexhatár hagyható el, a többi kell az adott tömbelem címének meghatározásához:

```
void ff (int it [] [20][50],  ..... )
{
    if (it [x1][x2][x3] >= 0) ....
    ....
}
```

Pl. itt `it[x1][x2][x3]` **címe :**

```
(char*) it + sizeof(int) * ( 20*50*x1 + 50*x2 + x3)
```

Kezdeti érték megadása soronként saját `{}` között, vagy sorfolytonosan is lehet:

```
int t2d [5][4] = { {11,12,13,14},      :  0. sor
                   {21,22},            :  1. sor eleje, majd 0
                   {},                 : HIBA: nem lehet üres
                   {41,42,43} };       :  3. sor eleje, az
                                          összes többi elem 0

float f2d [2][3] = {1.1, 2.2, 3.3, 4.4};  : sorfolytonosan,
                 = {{1.1, 2.2, 3.3},      : mintha ez lenne
                    {4.4} };              : a többi 0.0f
```

Tömb eleme bármi lehet, kivéve függvény,

de lehet függvényre mutató pointer, pl.:

```
double f1 (int,char);
double f2 (int,char);                 :  két függvény

double (*fpt[10]) (int,char) = { f1, &f2 };
```

`fpt` : **10 elemű, pointerekből álló tömb, melynek elemei `double` visszatérési értékű, `int` és `char` argumentum-típusú függvényekre mutat<u>hat</u>nak, az első kettő kapott kezdeti értéket**

Vegyük észre, hogy a függvény neve `()` nélkül a függvény címét jelenti, de ki is tehetjük az `&` cím-operátort.

Ugyanez másként:

```
typedef double f_t (int,char);        :függvény típus
f_t * fpt [10] = { f1, &f2 };         :pointerek tömbje
```

De: `double *fpt[10] (int,char) = { f1, f2 };`

HIBÁS: a `()` -nek, mint a függvényt jelentő operátornak magasabb a prioritása, mint a `*` -nak, azaz az indirekció jelének, így ez 10 elemű tömböt jelentene, melynek elemei olyan függvények (:ez a hibás), melyek `double`-re mutató pointert adnak vissza.

Nevek érvényességi köre, külön fordítás

Blokkban: { .... } definiált adatok:

- lokálisak a blokkra,
- elfedik a külső, azonos nevű dolgokat a blokk végéig,
- ha nem `static` (tehát az alapértelmezés szerinti `auto`): ilyen változónak a helyet a blokkba belépve foglalja, kilépve felszabadítja,
- ha `static`: program indulásakor foglal helyet, állít be kezdeti értéket, mely csak konstans kifejezés lehet.

Példa `static` adatra: függvény két string konkatenációjának előállítására az argumentumok megváltoztatása nélkül

```
char * SaferStrCat (char * s1, char * s2)
{   static char result [501];
    if (strlen(s1)+strlen(s2)>500) /* túl hosszú */
       return NULL;
    strcpy (result, s1);
    strcat (result, s2);
    return result;
} /* SaferStrCat() */
```

Használata pl:

```
char s1[200], * sp;
....
sp = SaferStrCat (s1, ″toldalék″);
printf (″%s″, sp);
....
```

Házi feladat (nehéz!): olyan változat, mely ismét meghívható saját eredményére is, pl.:

```
.... = SaferStrCat ( SaferStrCat (s1,s2),
                     SaferStrCat (s3,s4) );
```

Függvény formális paramétere:

- **lokális a függvényben, azaz függvény további részében látható, neve nem ütközhet a függvényben definiált v. deklarált más lokális névvel. Pl.:**

```
void csarda (int dudas, double citeras)
{    int dudas;                          :HIBA: két dudás!!!
     ...
     if (citeras == ugyes)
     {  double dudas;                    :OK.: lokális e blokkban
          ...
     }
     ...
}
```

File szinten, azaz legkülső szinten definiált változó, függvény:

- **file további részében használható,**
- **ha nem `static`: külön fordított modulokban (≡ file-okban) is látható (akár van előtte `extern`, akár nem)**
- **ha `static`: csak ebben a modulban látható**

Pl.:

`Egyik` modul	**`Masik`** modul	Magyarázat
`int adat = 0;` : ez definíció;	`extern int adat;` : ez deklaráció (⇒ tilos inicializálni)	ugyanaz; ez az ajánlott alak

`int i_tomb [100];`
: ez definíció;
	`extern`
	`int i_tomb [];`
	: ez deklaráció
		ugyanaz;
		csak a definícióban kell méret, de lehet a deklarációban is

`static` `char w [80];`	`static char w [80];`	ezek függetlenek
`double szint;`	`static double szint;`	ezek is függetlenek
`char Uzenet[20];`	`char Uzenet[20];`	HIBA: két definíció!!!

`void kiir (int d);`
:deklaráció
 	
`hasznalo (void)`
`{ int y;`
`...`
`kiir(y-2);`
`}`
	`void kiir (int x)`
	`{`
	`printf ("%d", x);`
	`}`
	: definíció
		ugyanaz
		
		`Egyik` -beli függvény hívja a **`Masik`** -beli **`kiir`** függvenyt

Külön fordítás: több forrásmodulból ≡ forrás-file-ból állóprogramok fordítása, összefűzése

A fordítóprogram és linker (≡ linkage editor) általában nem képes arra, hogy ellenőrizze a külön fordított modulokban definiált adatok és különösen a függvények fejeinek egyezését, ezért a következő szabályokat érdemes betartani:

1. **Minden külön fordított függvény deklarációját, azaz prototípusát meg kell adni pontosan egy header file-ban; ennek tipikusan `.h` a kiterjesztése.**
2. **Minden fordítási egységben, ahol ezt a függvényt hívjuk, `#include` -oljuk ezt a header file-t.**

   ⇒ nem lehet a header-rel ellentmondó hívás
3. **A külső függvény definíciója előtt is érdemes `#include`-olni ezt a header file-t.**

   ⇒ a függvény deklarációja és definíciója is konzisztens lesz, mert a fordító ellenőrzi a deklaráció és definíció egyezését
4. **A közös adatokat is meg kell adni header file-ban `extern` módosítóval.**

Egy példa:

file-1.h	file-2.h
`extern int Hibakod;`	`int Min (double t[], int s);`

file-1.c

```
#include ″file-1.h″
#include ″file-2.h″

int Hibakod = 0; /* def. */

int main()
{
   int n,p;
   double da[100];
   ....
   p = Min (da,n);
   if (Hibakod != 0) {
      fprintf (stderr,
           ″\n Hiba = %d\n″,
           Hibakod);
      exit(1);
   }
   ...
}
```

file-2.c

```
#include ″file-1.h″
#include ″file-2.h″

int Min (double t[],
         int s)
{
   int x;
   double min;

   if (s<=0) {
      Hibakod=1;
      return 0;
   }

   for (min=*t, x=1; x<s;
        x++)
      if (t[x]<min)
         min=t[x];
  Hibakod=0;
  return min;
}
```

+ az összefűzéshez link-nek, IDE-nek, cc-nek, vagy make-nek meg kell adni az összefűzendő modulok listáját, és esetleg függési viszonyait.

(lásd pl. `file-12.zip, thesaur.zip`**)**

struct : struktúra

≈ Pascal record, pl.:

```
struct sn { int    a, b;          : sn ≡ struktúra neve
            double dd;
          }  sv1, sv2;            : ilyen típusú változók

struct sn x1,x2;                  : további ilyen tip. változók

sn z1,z2;            : HIBA:  sn  nem típusnév, csak struktúranév

typedef struct { char m1;         : itt nem kell struktúranév
                 int t[5];
               } st_t;            :  st_t  típusnév lett
st_t y1, y2, *sp1;                : O.K.

typedef struct lancstrnev {       : ide kell str. név, hogy benne
   double d1;                       is  használható legyen,
   struct lancstrnev *kov;        : pl. így
} lancelem_t;                     : ez már típusnév

lancelem_t *elso, *utolso;        : a típusnevet használjuk
```

Struktúra elemeire való hivakozás:

Alakja: **struktúraváltozó . mezőnév**

Pl.:

```
sv1.a
sv2.dd
y1.m1
y2.t[3]
```

`*sp1.m1` : HIBA: `.` prioritása nagyobb, mint `*` prioritása

`(*sp1).m1 ≡ sp1->m1` :O.K., `->` a rövidebb alak

Dinamikus tárkezelés:

`malloc(), calloc(), realloc(), free()`

könyvtári függvényekkel.

`#include <alloc.h>` vagy
`#include <stdlib.h>` kell, ezekben:

Memóriafoglalás:

```
void * malloc (size_t s);
```

- ha tud, lefoglal `s` byte-os területet (az un. heap-ben),

- ennek címét adja vissza, ha sikerült

- terület tartalma meghatározatlan.

- ha nem tud, `NULL`-t ad vissza ⇒ mindig meg kell vizsgálni!

(de legalább nem hal meg a program).

`void *` **:definiálatlan típusú adatra mutató pointer: olyan pointer típus, amelynek bármilyen típusú ptr. értékül adható, de `void *` csak `void*` -nak, és nem hivatkozható az általa mutatott adat sem (hiszen mérete nem ismert), csak típus-konverzióval ≡ cast -olással. Pl.:**

```
void * vp;
char * cp;

vp = cp;              : O.K.
cp = vp;              : TILOS, de:
cp = (char*) vp;      : O.K.

*vp = 1;              :TILOS, nem tudjuk, mekkora *vp
*((int*)vp) = 1;      :O.K., de nem túl stílusos
```

`malloc` **használata pl.:**

```
int *dp, meret, x;
   ....
meret= .....;
   ....
dp = (int*) malloc (meret * sizeof(int));
if (NULL == dp)  {          /* lehetne  !dp is */
     fprintf (stderr, "\n Nincs elég memória \n");
     exit(1);
}

for (x=0;  x<meret;  x++) {
  dp[x]=....;           :mintha dinamikus méretű "tömb" lenne
   ...
 }
```

Másik memóriafoglaló fv.:

```
void *calloc (size_t nitems, size_t s);
```

- `nitems * s` **méretű területet foglal,**
- **ha sikerül, ezt nullázza,**
- **ha nem sikerül, NULL-t ad vissza.**

Memória felszabadítása:

```
void free (void * block);
```

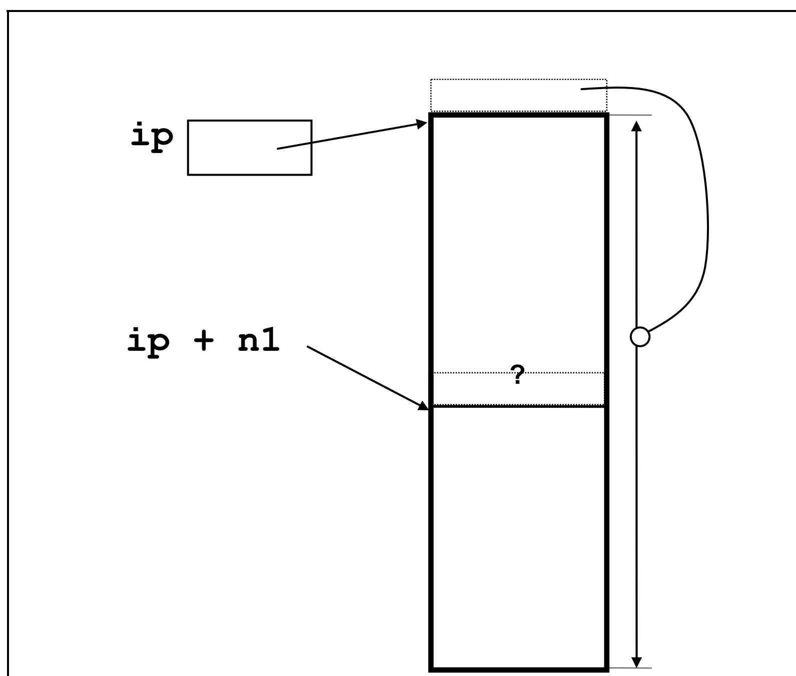
- `malloc, calloc, realloc` **által foglalt terület felszabadítása,**
- **csak az így foglalt terület elejét szabad neki átadni, közepét, stb. nem!**
- **méretet** `malloc` **(stb.) elteszi neki (célszrűen a terület elé), így azt ismeri, nem kell külön megadni** `free`**-nek**

Miért nem szabad a lefoglalt terület végét `free` -vel felszabadítani?

Pl.

```
int * ip, n1;
...
ip = (int*) malloc (100 * sizeof(int));
...
n1 = ...;                                    < 100
free (ip+n1);              :  maradék rész felszabadítása
```

Ami történne:



Dinamikusan foglalt memória méretének megváltoztatása:

```
void* realloc (void* régi_cím,  size_t új_méret);
```

- dinamikusan foglalt terület csökkentésére, vagy megnyújtására
- megnyújtásnál a régit követő területet is lefoglalja, ha lehet, ha pedig az foglalt, új helyen foglal helyet, a régi tartalmát oda másolja, majd a régi területet felszabadítja;

  ha nincs elég szabad hely a heap-ben: a régi megmarad, `NULL` -t ad vissza
- `régi_cím` lehet `NULL` is, ekkor nincs régi terület, de foglal újat

Pl. függvény a heap-ben lefoglalható legnagyobb terület méretének meghatározására:

```
size_t HeapSize (void) {
  void   *p;                  /* ==> allocated area */
  size_t  high=1024, low=0,   /* limits of length */
          size_t_max = -1;    /* max. of size_t */

  while ( (p = malloc (high)) != NULL ) {  /* double */
    free(p);
    low = high;
    high = size_t_max/2 < high ? size_t_max : high*2;
                            /* high*2 would overflow */
  }  /* now: maximal size is in low .. high interval */

  while (low<high) {             /* bi-section method */
    size_t mid = (low+1)/2+high/2;    /* middle,
                               low+high may overflow */
    if ( (p = malloc (mid)) != NULL ) {   /* success */
      low=mid+1;
      free(p);
    }
    else high=mid-1;
  }
  return high;
}  /* HeapSize () */
```

Házi feladat / 1: Alakítsa át az előbbi függvényt úgy, hogy akkor is jól működjön, ha `size_t` nem elég nagy ahhoz, hogy a heap méretének kétszeresét is tárolni tudja (pl. DOS-ban)!

Házi feladat / 2: miért nem `realloc`-al csináltuk?

File : stream kezelés

`#include <stdio.h>` **kell, ebben:**

`FILE` **:egy típusnév, melyre mutató pointerrel használhatunk file-okat, pl.:**

```
FILE * befile, * kifele;
```

File megnyitása:

```
FILE * fopen (char * filenév,  char * mode);
```

`mode` **első karaktere lehet:**

- `r` **:read = olvasásra nyitja meg**
- `w` **:write = írásra létrehozza**
- `a` **:append = hozzáírásra nyitja meg, a file eddigi tartalma végére pozicionálva (konkatenáció)**

második karaktere lehet:

- `+` **:munka-file (írható és olvasható is)**

majd ezek után lehet valamelyik:

- `b` **:bináris file**
- `t` **:text, azaz szöveges file**

Ha nem sikerül megnyitni, `NULL` **-al tér vissza, pl. rossz file-név, mode,**

- `r` **: a file nem létezik**
- `w` **: a hordozó írásvédett, vagy tele van**

Mindig ellenőrizni kel !!!

Pl. az `adataim.dat` **file-t nyitja meg olvasásra, szövegfile-ként:**

```
if ((befile = fopen ("adataim.dat", "rt")) == NULL)
    fprintf (stderr, "\nNem tudom megnyitni az"
             " \"adataim.dat\" file-t olvasásra\n");
```

File lezárása:

= buffer kiírása, directory update

```
int fclose (FILE * fp);
```

= `0` ha OK,

= `EOF` ha hiba történt. (`EOF`: makró, spec. értéket ad meg, pl. –1)

Igaz ugyan, hogy a program befejezésekor automatikusan le kell.
hogy záródjanak a file-ok, de nem szokás a rendszerre bízni!

Hiba vizsgálatára használhatók:

```
int feof (FILE * stream);
```

`!= 0`, ha end-of-file jött

```
int ferror (FILE *stream);
```

`!= 0` ha hiba történt, hibakód: `extern int errno`

Szabványos stream-ek:

szövegesek, program indításakor már meg vannak nyitva:

`stdin` : szabványos bemenet (átirányítása: `<fnév`)

`stdout` :szabványos kimenet (átirányítása: `>fnév`)

`stderr` :szabványos hiba-kimenet (átirányítása: `2>fnév`)

Pl. program, mely 1. paramétereként megadott szövegfile-t kiírja a 2. paraméterként kapott nevű file-ba:

```
#include <stdio.h>

int main (int    argc,       /* paraméterek száma +1 */
          char * argv[])     /* paraméterstringek */
{
  int c;
  FILE * infile, * outfile;

  if (argc != 3) {
    fprintf (stderr, "\nHasználata: %s  "
                     "bem.file  kim.file\n", argv[0]);
    return 1;
  }

  if ( (infile = fopen (argv[1], "rt")) == NULL ) {
    fprintf(stderr, "\nNem tudom olvasásra megnyitni:"
                    "  %s\n", argv[1]);
    return 2;
  }

  if ( (outfile = fopen(argv[2],"wt")) == NULL) {
    fprintf (stderr, "\nNem tudom írásra megnyitni:"
                     "  %s\n", argv[2]);
    return 3;
  }

  while ( (c=getc(infile)) != EOF)  putc(c,outfile);

  fclose(infile);  fclose(outfile);
} /* main() */
```

Bináris, azaz formázás nélküli beolvasás és kiírás:

```
size_t fwrite (const void * innen,
               size_t       elemhossz,
               size_t       elemszám,
               FILE       * stream);
```

- kiír az `innen` által mutatott területről
- `elemszám` darab
- `elemhossz` méretű [byte] adatot a
- `stream` file-ba,
- visszaadja a sikeresen kiírt elemek számát, (nem byte-számot), ami hiba esetén `elemszám` -nál kevesebb.

```
size_t fread (void   * ide,
              size_t   elemhossz,
              size_t   elemszám,
              FILE   * stream);
```

- beolvas az `ide` által mutatott területre
- `elemszám` darab
- `elemhossz` méretű [byte] adatot a
- `stream` file-ból,
- visszaadja a sikeresen beolvasott elemek számát, (nem byte-számot), ami `elemszám` -nál kevesebb is lehet, ha közben a file végére ért, hiba esetén 0-t ad.

Ezek szekvenciálisan (sorrendben) írnak/olvasnak.

Közvetlen pozícionálás file-ban:

```
int fseek (FILE * stream,  long offset,  int whence);
```

`offset` :relatív cím [byte]

`whence` :mihez relatív:

- `SEEK_SET` : file elejéhez képest
- `SEEK_CUR` : jelenlegi pozícióhoz k.
- `SEEK_END` : file végéhez képest

= 0 : O.K.

!= 0 : hiba, pl. file nincs megnyitva

A további `fread` ill. `fwrite` műveletek az így beállított ponttól dolgoznak.

Ha `fseek`-el a file vége utánra állunk, majd írunk, a kimaradt részre bináris nullák kerülnek.

Pillanatnyi pozíció, azaz offset leolvasása file elejéhez képest, ill. hiba setén `-1L` :

```
long ftell (FILE * stream);
```

Megjegyzés: egyes rendszerekben `long` nem elég hosszú a file méretének megadásához, ekkor `ofs_t` az erre definiált típus.

Formázott kiírás szöveg (text) file-ba

Az eddigi műveletek egy memóriaterületet mozgattak bináris alakban, azaz ember számára nem olvashatóan.

```
 printf (            char * control, ...);     :stdout-ra
fprintf (FILE * fp, char * control, ...);     :fp file-ba
sprintf (char * sp, char * control, ...);     :sp bufferbe
```

A control string karaktereit változtatás nélkül kiírja, kivéve a `%` jellel kezdődő nyomtatásvezérlő mezőket, ezeknek rendre egy-egy (esetleg 2, 3) további paraméter kell a `...` helyen.

Ezek meglétét és helyességét a compiler ill. a függvény nem tudja ellenőrizni!

A nyomtatásvezérlő mező alakja:

`% [flag-ek] [szélesség] [.pontosság] [h|l|L] típus`

Megadja, hogy milyen típusú a megfelelő argumentum, illetve, hogy azt milyen alakban akarjuk kiírni. (A [..] itt most elhagyható részt jelöl.)

Ha a `%` -ot követő karakter nem ennek megfelelő, akkor az illető karakter nyomtatódik ki, pl. `%%` a `%` jelet nyomtatja.

`flag-ek:`

- `-` : **az argumentum balra igazítását végzi; ha nincs, jobbra igazít**
- `+` : **+ előjelet is mindig kiírja, egyébként csak - -t.**

`szélesség` : **minimális mezőszélesség: decimális szám, vagy `*,` ekkor a mezőszélességet a következő `int` argumentum adja meg**

`.` : **mezőszélesseg és a pontosság elválasztása**

`pontosság` : `float` és `double` esetében a tizedes jegyek száma, karakterlánc esetén a max. karakterszám;
ha `*`, akkor a következő `int` argumentum adja

`h|l|L` : hossz módosító; a kiírandó érték:

```
h :short int
l :long int
L :long double
```

`típus` : érték típusa, a kiírás formája

`d` :	`int`	:decimális
`i` :	`int`	:decimális
`o` :	`int`	:előjel nélküli oktális
`x` :	`int`	:előjel nélküli hexadecimális
`u` :	`unsigned`	:előjel nélküli decimális
`c` :	`char`	:karakter
`s` :	`char *`	:karakterlánc
`e` :	`double`	:kitevős alak: `m.nnnnnne±kk`
`E` :	`double`	:kitevős alak: `m.nnnnnnE±kk`
`f` :	`double`	:fixpontos alak: `-mmm.nnnnn`
`g` :	`double`	:`e` és `f` közül a pontosabb
`G` :	`double`	:`E` és `f` közül a pontosabb

Pl.:

```
  int x=123, h=7;
  long lint=123456;
  double d=1e-8;

 printf (
   "\n First =%d  2nd:%3cc  >%*li<    (%10G)  \"%s\" ",
             x,      'q',   h,lint,     d,    "This");
```

Eredmény:

```
First =123  2nd:  qc  > 123456<    (     1E-08)  "This"
```

Formázott beolvasás szöveg (text) file-ból

```
int  scanf (            char * control, ... );  :stdin-ről
int fscanf (FILE *fp,   char * control, ... );  :file-ból
int sscanf (char *str,  char * control, ... );  :string-ből
```

A control, azaz formátumvezérlő stringgel megegyező szöveget kell találnia, egyébként leáll, kivéve ha a vezérlő stringben van:

közök (szóköz, tabulátor, újsor): bemenetben átugrik minden közt;

`%` -kal kezdődő vezérlőmezők: adott típusú értékeket olvas és tárol, ezeknek rendre egy-egy további memóriacím kell a `...` helyen, kivéve tárolás elnyomásakor (pl. `%*c`) !

!!! AZ AKTUÁLIS ... ARGUMENTUMOKNAK CÍMEKNEK (MUTATÓKNAK) KELL LENNIÜK !!!

Akkor áll le, ha:

- **a bemeneti szöveg nem egyezik a formátum szöveggel,**
- **a teljes control stringet feldolgozta,**
- **a bemeneti adatok végére ért (EOF, v. string vége),**
- **valamelyiket nem tudta beolvasni, mert hibás volt a beolvasandó adat formátuma.**

Visszatérési értéke: hány értéket olvasott be és tárolt el az argumentumokban sikeresen.

Érték előtti közöket általában átlépi, kivéve `%c` és `%[....]`

A control mező felépítése (a [..] közötti rész elhagyható, | választható):

% [*] [szélesség] [h | l | L] típus

`%` **:vezérlőmező kezdete**

`*` **:tárolás elnyomása: ezen formátum-előírás szerinti szöveget átolvassa a file-ból, de az értékét nem tárolja el, így ...`scanf` visszatérési értékébe sem számlálja bele**

szélesség: decimális szám: max. mezőszélesség

`l` **:adathossz módosító, a megfelelő argumentum:**

`int*` **helyett** `long*`

`float*` **helyett** `double*`

`L` **:adathossz módosító:** `float*` **helyett** `long double*`

típus: konverzió típusa:

(Elöl (`l`) ill. (`L`) azt jelenti, hogy lehet ilyen hossz-módosító)

(`l`) `d` : `int*` (`long*`) **:decimális egész számot vár**

(`l`) `o` : `int*` (`long*`) **: oktális egész számot vár**

(`l`) `x` : `int*` (`long*`) **: hexadecimális egész számot vár**

`h` : `short*` **: short egész számot vár**

(`l|L`) `f` : `float*` (`double*` **vagy** `long double`) **: lebegőpontos számot vár**

`c` : `char` **: egyetlen karaktert vár, az üres karaktert (szóköz, újsor, tabulátor) is beolvassa. Ha át akarjuk ugorni az első értékes karakter előtti üres karaktereket, akkor** `" %c"` **kell, azaz a** `%c` **előtt kell legalább egy szóköz v. tabulátor.**

`s` : `char*` : egyetlen szót vár, előtte az üres karaktereket átlépi, első üres karakterig olvas, lezáró `'\0'` karaktert is eltárolja

`[`karakterek`]` : `char*` :stringet olvas be, amíg ilyen karakterek jönnek be, a `[` és `]` között a karakterek megengedett halmaza adandó meg. Pl.:

`[-+0-9]` :előjelek, dec. számjegyek, bármely sorrendben, pl.

`12-43+0.1` (aláhúzva: `12-43+0`)

esetén az aláhúzott részt olvassa be

`[a-zA-Z]` :betűk

`[^ \n\t]` :minden, csak üres karakter nem, mert `^` a komplemens halmazt jelenti

Pl. `%*[^:]%99[^\n]` :átugrik mindent ':'-ig, majd sor végéig beolvas mindent, de legfeljebb 99 karaktert

```
/*==== NeptNevs.c  :  Neptun nevsor feldolgozása ====

  Bemeneti adatok alakja pl.:

Szgép.lab.II.(BMEVIEE1234-01) kurzus adatai - Hallgatók
1.

Sorsz.   Kód Hallgató neve    Jelentkezési dátum

2    JR8DBC   Domokos Péter    1999.01.19 15:11:20
11   YRX1X6   Huszár Csaba Pál 1999.02.19 13:22:34
8    JYWJ38   Jónás Balázs Árpád    1999.01.19 18:29:45
3    W4Q1LR   Király Márton György 1999.01.19 15:16:00
6    Z0N8AH   Mihályi Antal    1999.01.19 15:23:34
...
Tudjuk hogy a mezők között tabulátorok vannak.

Eredmények pl.:

Domokos Péter          JR8DBC
Huszár Csaba Pál       YRX1X6
Jónás Balázs Árpád     JYWJ38
Király Márton György   W4Q1LR
Mihályi Antal          Z0N8AH
...
=====================================================*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>   /* qsort() ebben van */

#define max_nevhossz 50
#define max_kodhossz 6

typedef struct {  char nev [max_nevhossz+1];
                  char kod [max_kodhossz+1];
               } hallgato_t;

/*--- Két bejegyzés összehasonlítása : qsort()-hoz --*/
int hasonlit (const void * bal,  const void * jobb)
{
   return strcmp (((hallgato_t*)bal)->nev,
                  ((hallgato_t*)jobb)->nev);
}
```

```
main ()
{
  #define max_letszam 350
  hallgato_t  nevsor [max_letszam];

  int olvasott;          /* beolvasott értékek száma */
  int letszam;           /* tényleges létszám */
  int x, nevhossz;       /* index, max. névhossz */

  for (letszam = nevhossz = 0;  letszam < max_letszam;
) {
    olvasott = scanf ("%*d %6s %50[^\t] %*[^\n]\n",
      /* sorszám átlépépse, Neptum kód beolv.,
         közök átlépése, név olvasása tabulátorig,
         sor maradék részének átugrása */
             & nevsor[letszam].kod,
             & nevsor[letszam].nev);
    if (2 == olvasott) {       /* sikeres az olvasás */
      x = strlen (nevsor[letszam].nev);    /* max
                                    névhossz meghat. */
      if (x > nevhossz) nevhossz = x;
      letszam++;
    }
    else scanf ("%*[^\n]\n");   /* sor további részének
                                   átugrása */
    if (feof(stdin)) break;
  } /* for */

  if (max_letszam == letszam) {
     fprintf (stdout, "\n Túl sok a hallgató\n");
     return 0;
  }

  qsort (nevsor, letszam, sizeof(nevsor[0]), hasonlit);

  for (x=0;  x<letszam;  x++)
    printf (" %-*s  %s\n", nevhossz+2, nevsor[x].nev,
                           nevsor[x].kod);

  return 0;
}
```

Hibakód rendszerhívások (pl. I/O) után

```
<errno.h>
<stddef.h>
<stdlib.h>

    extern int errno;
```

== 0 : rendben
!= 0 : hiba

Hibaüzenet stringek az egyes hibakódokhoz:

```
<stdlib.h>

    char * sys_errlist [];
```

Hibaüzenetek kiírása `stderr` file-ba:

```
<stdio.h>

    void perror (const char *s);
```

ami így működik:

```
  fprintf (stderr, "%s : %s\n", s, sys_errlist[errno]);
```

Változó ill. definiálatlan számú és típusú függvény-paraméterek

:adott függvény egy-egy helyen más-más típusú v. számú paraméterrel is hívható.

A függvényfej alakja pl.:

```
v_típus fuggvénynév (char *par1, ... )
                                 ^^^  ez itt pontosan 3 pont
```

Szabályok:

- Legalább 1 'fix' paraméter van, de lehet több is.
- A visszatérési érték és a fix paraméter(ek) típusa bármi lehet
- A `...` csak `,` után lehet, csak utolsó lehet.
- Az első argumentum akármi lehet, de arra szokás használni, hogy valahogyan megadja a továbbiak számát és típusát.
- Az aktuális paraméterek értékén az un. szokásos konverziókat hajtja végre a gép:

```
char, short     ⇒ int
float           ⇒ double
unsigned char   ⇒ int, vagy ha abban nem fér el,
                  unsigned int
unsigned short  ⇒  - ″ -
T tömbje        ⇒ T *          ( T egy típus)
T fv (.....)    ⇒ T (*fp) (.....)
```

A változó argumentumlista feldolgozásához a `<stdarg.h>` **-ban rendelkezésre áll:**

`typedef` valami `va_list;` :`va_list` **típus definíciója; ilyen típusú munkaváltozó kell az aktuális argumentumok elővételéhez, pl.:**

```
va_list argp;
```

`void va_start(ptr, utfixarg)` **:makró a** `ptr va_list` **típusú munkaváltozót inicializálja,** `utfixarg` **az utolsó fix argumentum neve, pl.:**

```
va_start(argp, UtolsóFixArg);
```

`va_arg(ptr, típus)` **:makró a következő argumentum elérésére, pl.:**

```
int x; ....;
x = va_arg(argp, int);
```

`va_end(ptr)` **:makró az argumentumista feldolgozásáanak befejezéshez, pl.:**

```
va_end (argp);
```

1. példa: Függvény tetszőleges számú és sorrendű `int`, `short` és `char` érték minimumának megkeresésére:

```
int MinNInt (int parno, ...)
{
    va_list ap;          /* work variable to access the
                            arguments */
    int min;             /* present minimum */

    va_start (ap, parno);    /* init. argument ptr.  */
    min = va_arg (ap,int);   /* value of first arg.  */
    while (--parno>0)        /* there are more args. */
    {  int nextarg = va_arg(ap,int);     /* get next */
       if (nextarg < min)  min = nextarg;
    }
    va_end(ap);          /* end of argument processing */

    return min;
}
```

Hívása pl:

```
    int x1,x2,x3;   short s;  char ch1, ch2;
    ......
    x1 = MinNInt (5, x2,x3,ch1,ch2,s);
    ......
    valami = MinNInt (3, ch2,s,x3);
```

Egy kérdés:

Miért nem jó pl. ez:

```
    ...
    while (--parno>0)
       if (va_arg(ap,int) < min)
           min = va_arg(ap,int);
    ...
```

2. példa: Készítendő egy függvény, amely `int, char, double, string` és `long int` típusú értékeket tud kiírni `stdout`-ra, akárhányat, akármilyen sorrendben.

Megoldás: az 1. argumentumnak egy stringet használunk, melynek karakterei rendre megadják a további argumentumok típusait:

```
i ⇒ int      c ⇒ char     s ⇒ string
d ⇒ double   l ⇒ long
```

```
/*== StadArgP.c  :  Példa <stadarg.h> használatára ==*/

#include <stdarg.h>
#include <stdio.h>

int main()    /* a hívó függvény */
{
  char ch='d';
  long lv = 88888888;

  void printargs (char *argtypep, ...);   /* deklar. */

     /* 1. hívás alakját írja ki: */
     printf("\n  printargs (\"id\", 1, 1.1);\n");

     printargs ("id", 1, 1.1);           /*--- 1. hívás ---
*/

     /* 2. hívás alakját írja ki: */
    printf("\n  printargs (\"icddcsl\", 222, 'c', "
           "444.444, 55.55F,ch,"
           "\n               \"7. param\", lv);\n");

                                        /*--- 2. hívás ---
                                    */
     printargs ("icddcsl",   /* :argumentum-típusok   */
                222,          /* 2. arg.: int            */
                'c',          /* 3. arg.: char->int      */
                444.444,      /* 4. arg.: double         */
                55.55F,       /* 5. arg.: float->double*/
                ch,           /* 6. arg.: char -> int  */
                "7. param",  /* 7. arg.: string         */
                lv);          /* 8. arg.: long int       */
    return 0;
}
```

```
/*---- A változó argumentumú függvény definíciója ---*/

void printargs (char *argtypep, ...)
{ va_list ap;            /* munkaváltozó a lépkedéshez */
  int ctr;               /* argumentum számláló */

  va_start (ap, argtypep);    /* kezdeti beállítások */
  for (ctr=2;  *argtypep;  argtypep++, ctr++) {
    printf ("\n   arg[%d]: ", ctr);       /* sorszám */
    switch (*argtypep) {
    case 'i':  printf("int     : %d",  va_arg(ap,int));
               break;
    case 'c':  printf("char    : %c",
                              (char)va_arg(ap,int));
               break;
    case 's':  printf("string  : %s",
                                 va_arg(ap, char*));
               break;
    case 'd':  printf("double  : %f",
                                va_arg(ap, double));
               break;
    case 'l':  printf("long int: %ld",
                             va_arg(ap, long int));
               break;
    }
  }
  va_end(ap);            /* arg. lista feldolg. vége */
  printf("\n");
}                         /* printargs() vége */
```

Amit ez a program kiír:

```
printargs ("id", 1, 1.1);
         arg[2]: int  : 1
         arg[3]: double  :1.100000

printargs ("icddcsl", 222, 'c', 444.444, 55.55F, ch,
           "7. param", lv);
         arg[2]: int  : 222
         arg[3]: char : c
         arg[4]: double  : 444.444000
         arg[5]: double  : 55.549999
         arg[6]: char : d
         arg[7]: string  :7. param
```

```
arg[8]: long int: 88888888
```

Konstans objektumok: `const`

Pl.:

```
const int c1;     :konstans int, közvetlenül nem változtatható meg
                    az értéke, de kezdeti érték megadható:
const int c2=33;                : O.K.
  c1=0;   c2=33;                :HIBÁS mindkettő

const float * pcf;        pointer, mely konstans float-ra mutathat
float f, *fp;
pcf=&f;                   O.K.: a pcf pointer beállítható
*pcf=2.2;                 HIBA: a mutatott érték const
fp=pcf;                   HIBA: fp nem konstansra mutat
fp=(float*) pcf;          O.K., de így felülírható egy konstans:
*fp=-1.1;                  formálisan jó, de ...

double d, *dp=&d;
double * const dcp;        double-re mutató konstans pointer
dcp=&d;                    HIBA: dcp konstans
dp=dcp;                    formailag O.K., de hova mutat dcp?
*dcp=-3.3;                 formailag O.K., de hova mutat dcp?
```

⇒ Konstans objektum inicializálás nélkül mire jó???

Konstans formális függvény-paraméter: jelzi, hogy a fv. nem változtatja meg az argumentumot, pl.:

```
char * strcpy (char * dest,  const char  * src);
```

Felsorolás típus (enumeration type)

Mint Pascal felsorolt típus, de valamelyik egésszel (`short / int / long`) azonos típusú lesz, hogy melyikkel, az megvalósításfüggő. Pl.:

```
enum napok_e {He, Ke, Sze, Cs, Pe, Szo, Va}
     ma, holnap;
```

Értékei: 0, 1, ... 6

További ilyen változók definiálása:

```
enum napok_e tegnap, napok[10];    /* kell az enum ! */
```

Egy függvény ilyen értékekkel:

```
enum napok_e Holnap (enum napok_e ma)
{  return ma == Va ? He : ma+1;  }
```

Némelyik fordító int ⇒ enum konverzóhoz cast-olást vár, pl:

```
{  return ma == Va ? He : (enum napok_e)(ma+1);  }
```

Érték is megadható, ekkor a következő alapértelmezésben az előző+1 értéket kapja, pl.:

```
enum ulohelyek { Pista=1, Julcsa, Pali=5, Panni,
                 Terez, Tibor=9 };
```

Ekkor: `Pista=1, Julcsa=2, Pali=5, Panni=6, Terez=7, Tibor=9`

Ha típust akarunk, akkor itt is `typedef` kell, pl.:

```
typedef enum tennis_points_t {love, fiveteen,
     thirty, fourty, game} tennis_points_t;
```

Ezzel egy függvény:

```
    /* == 0      : game is not over,
       >0 / <0   : game over, player 1 / 2 won */

int GameWon (tennis_points_t pt1,
             tennis_points_t pt2)
{  if ((pt1==game || pt2 == game) &&
       abs(pt1-pt2) >= 2)   return pt1-pt2;
   return 0;
}
```

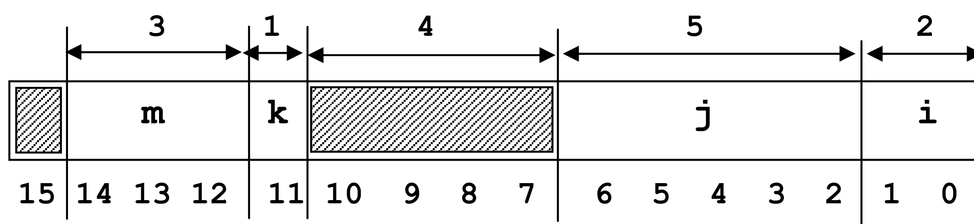
Bit-mezők

- Struktúra elemeinek hosszát bit-ben adjuk meg
- az elemek egészek (int, unsigned, char long, ...) lehetnek
- `int` típusú bitmező akár előjeleset, akár előjel nélkülit jelenthet (mint char típus)
- egy bit-mező maximális hossza megvalósításfüggő
- a memóriában az elemek nem csak byte-határon, hanem „pakoltan” helyezked(het)nek el
- a tényleges bit-sorrend, a kihagyások léte és hossza megvalósításfüggő
- 0 hossz tárolási egység (pl. szó-) határra igazítást jelent

Példa:

```
struct mystruct {
  int       i : 2;      /* 2 bites */
  unsigned  j : 5;
  int         : 4;      /* 4 bites, nincs neve */
  int       k : 1;
  unsigned  m : 3;
} a, b = {0,0,0,0};     /* kezdeti érték */
```

BC 5.02 dokumentáció szerinti szerkezete:



Ha lefuttajuk ezt (lásd `BitField.c`) :

```
printf ("\n sizeof(a) = %ld\n", (long) sizeof (a));
a=b;
printf ("\n a.i:  "); a.i=-1;    DUMP (a); a=b;
printf ("\n a.j:  "); a.j=-1;    DUMP (a); a=b;
printf ("\n a.k:  "); a.k=-1;    DUMP (a); a=b;
printf ("\n a.m:  "); a.m=-1;    DUMP (a);
```

BC 3.1 alatt ezt kapjuk:

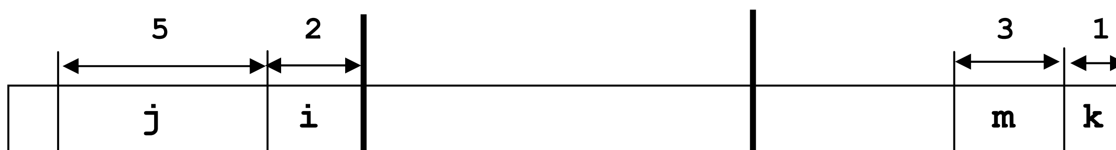
```
sizeof(a) = 2
a.i:  03 00
a.j:  7c 00
a.k:  00 08
a.m:  00 70
```

ami megfelel a fenti dokumentációnak.

BC 5.02 alatt viszont ezt kapjuk:

```
sizeof(a) = 3
a.i:  03 00 00
a.j:  7c 00 00
a.k:  00 00 01
a.m:  00 00 0e
```

Az ennek megfelelő szerkezet:



⇒ a megvalósítást ellenőrizni kell, akár a dokumentáció is „tévedhet”

⇒ a tényleges elhelyezést befolyásolhatja a byte-sorrend is, mint Intel processzoroknál

⇒ megbízható és hordozható tárolást csak bitenkénti beállítással érhetünk el (lásd: Eratosth.c)

Lásd még: BitField.c

Union

Alakja hasonló a struct-éhoz.

Példa:

```
union uni_u {
  double        r;
  unsigned char ch[10];
} v1, v2;
```

- **az elemeit azonos területen helyezi el**
- **minden eleme azonos címen kezdődik, ami az egész union változó címe**
- **az union hossza a legnagyobb elem hossza + esetleg kiegészítés megfelelő határra, hogy a teljes hossz az union bármely eleme igazításai határának többszöröse legyen.**
  Pl. ha `double` 8 byte-os határra kell, hogy kerüljön, akkor a fenti union 16 byte hosszú lesz.
- **csak azt az elemét lehet használni, amelyik utoljára értéket kapott**

Felhasználása:

- **egymást kizáró változatok tárolása azonos területen**
- **adott terület más típusként való felhasználása**
- **„veszélyes trükkök”**

Példa: valós szám belső ábrázolásának felderítése: `RealBits.c`